(2)

The Data Structure Accelerator Architecture

Richard Zippel*

TR 91-1256
December 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Naval Research Contract N00014-88-K-0591, the National Science Foundation through grant DMC-86-17355 and the Office of Naval Research through contract N00014-89-J-1946.

The Data Structure Accelerator Architecture

Richard Zippel*
Department of Computer Science
Cornell University
Ithaca, NY 14863
rz@cs.cornell.edu

December 17, 1991

Abstract

We present a fine grained, massively parallel SIMD architecture called the *data structure accelerator* and demonstrate its use in a number of problems in computational geometry. This architecture is extremely dense and highly scalable. Systems of 10^6 processing elements can be feasibly embedded in work stations. We advocate that this architecture be used in tandem with conventional single sequence machines and with small scale, shared memory multiprocessors. We present a language for programming such heterogeneous systems that smoothly encorporates the SIMD instructions of the data structure accelerator with conventional single sequence code.[1]

1 Introduction

There has been a significant body of work on single instruction, multiple data (SIMD) computer architectures in the past. This work ranges from the MPP machine developed at Goodyear [4] in the 1970's to the Connection Machine today [3]. These machines are generally viewed (and sometimes even advertised) as large "supercomputers." They are intended to be used for large problems that are not practical on smaller machines. However, these SIMD machines are not uniformly better than conventional processing elements or the new generation of MIMD processors for all problems. Furthermore, it is rarely cost effective to couple the large SIMD machines with other types of processing elements, so that a combined, heterogeneous machine can be applied to a problem; each component performing those computations at which it is best.

The existing SIMD machines have a relatively modest number of processing elements (the Connection Machine can handle up about $10^{4.8}$), which is compensated for with a sophisticated interconnection network. As a consequence, the chunks of computation performed on a SIMD machine tend to be quite large and there is relatively little interaction

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-88-K-0591, the National Science Foundation through grant DMC-86-17355 and the Office of Naval Research through contract N00014-89-J-1946.

[1]This is an expanded version of a paper that was presented at the *Jerusalem Conference on Information Technology, October 1990*

between the SIMD machine and its host(s) during the computation. This reinforces the division between SIMD computations and more conventional approaches.

We feel this division is counterproductive. To illustrate this present a number of algorithms in Section 4 that use a mixture of SIMD and SISD constructs to achieve high performance. Interestingly, these algorithms are quite simple compared the optimal algorithms for single sequence machines, and yet perform substantially better. The problems these techniques solve arise as small parts of much larger problems that are quite difficult to parallelize using SIMD techniques. It would be wildly impractical to devote a Connection Machine their resolution in most cases.

In this paper we suggest that the real role for SIMD architectures is not as "stand-alone supercomputers," but as an integral component of a heterogeneous machine consisting of both SIMD and SISD (or MIMD) components—each component responsible for the portions of a computation at which they are most effective. Often this means managing large, memory resident data structures, or performing simple operations on large blocks of data. Thus a natural way to merge a SIMD architecture with a conventional approach is to make the SIMD processing elements part of the memory system of more conventional single processor or MIMD architectures.

Thus we argue for simple SIMD architectures that can be built relatively cheaply and with very high density. We are interested developing systems with upwards of 10^6 processing elements which can be used in personal work stations and upwards of 10^8 for "supercomputing applications." The individual elements of such SIMD architectures must be quite simple to have this density and their interconnect must also be simple to allow for scalability and the size system we are interested. We have developed a class of SIMD architectures that meets these criteria which we call *Data Structure Accelerators* (DSA).[2] In Section 2 we present their organization in detail.

One of the problems with dealing with heterogeneous architectures is the difficulty of expressing algorithms clearly and succinctly. In Section 3 we describe a few natural extensions that could be made to an algebraic language like C that simplify the description of data parallel computations. This extended language is unique in that it allows one to express cooperative algorithms that are executed on a heterogeneous computer consisting of both a SIMD component and conventional single sequence component.

Having this type of computation cheaply available affects the type of algorithms that are used. Many of the complicated algorithms developed for searching and managing data structures are no longer necessary because simple DSA managed data structures can be used where all of their elements are handled in parallel. Some examples of this are given in Section 4, where discuss a few problems in computation geometry.

2 The Data Structure Accelerator Architecture

The Data Structure Accelerator (DSA) is a class of SIMD architectures that is extremely dense, easily scalable and has been optimized to efficiently perform functions that are difficult for conventional processing systems. The DSA's processing elements (PE's) are suffi-

[2] Earlier versions of this work at MIT referred to this architecture as a Database Accelerator. Since this effort is directed towards "in memory" databases our original choice of names was somewhat misleading. To correct this we have chosen the name *data structure accelerator*, which is more suggestive.

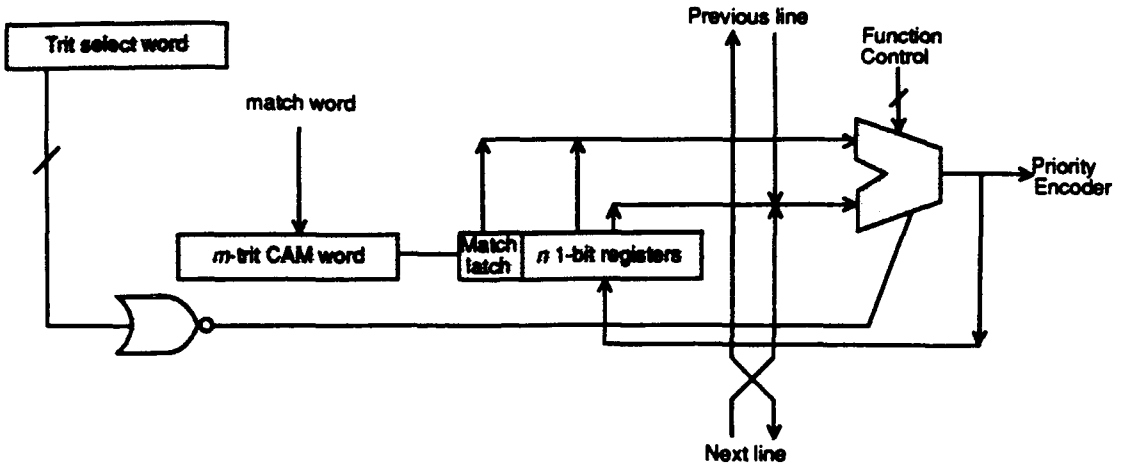

Figure 1: MIT Database Accelerator Architecture

ciently compact that machines with upwards of 10^7 processing elements are feasible—well into the massively parallel regime.

The processing elements are connected in a low dimension, rectangular grid. This type of interconnect is much cheaper, and can be scaled up more easily than the boolean n-cube network used by machines like the Connection Machine. Although this makes a few problems impractical, we feel the improved scale and cost of the resulting system worthwhile. For many applications a one dimensional interconnect is sufficient. For some problems in vision and computational geometry two dimensional interconnects are useful. In principle, higher dimensional interconnects could be used, but their impact on pin count would severely compromise our effort to build *massively* parallel systems. For now we have only considered one and two dimensional data structure accelerators.

To keep the processing elements small we have decided to restrict them to being a single bit wide. This minimizes the number of types of operations that need to be incorporated at each PE, but remains sufficiently general to be used for most applications. Because the DSA is often used to manage large tables, we include content addressable memory in the (virtual) DSA architecture. An example of such an element of a one dimensional linear array is shown Figure 1. Each processing element is called a *line*. A line contains some amount of content addressable memory (CAM) and random access memory (RAM). A particular data structure accelerator is characterized by four parameters: the number of lines in the DSA (ℓ), the dimensionality of the interconnection of their cells, the number of bits of CAM (m) and the number of bits of RAM (n). The *SelectWord* is shared by all the lines of the DSA.

When discussing algorithms that use the DSA, the dimensionality of the DSA is understood (it is usually one) and the number of lines is not important. However, the size of the CAM and RAM structures usually is important. Thus, we say that a data structure accelerator has *parameters* (m, n) when each line has m bits of CAM and n bits of RAM. The amount of CAM and RAM associated with each line of a DSA can be optimized for

different applications and can range from systems that only have CAM to those that only have RAM. This is discussed in more detail in Section 2.2. The Smart Memories Project at MIT has built a 64 line (32,4) DSA chip [6, 8], a 256 line (32,4) DSA chip [1] and a board that implements a 4096 line (32,4) DSA. With current technology, devices with thousands of lines are practical and systems in the range of 10^6 to 10^8 PE's are feasible.

Even though a particular set of parameters must be chosen when building a DSA system, the user can simulate a DSA with a different set of parameters at surprisingly little cost. This is demonstrated in Section 2.1. This feature is one of the most important differences between the DSA architecture and previous CAM approaches.

The data structure accelerator makes use of three valued logic for two purposes: They are used to indicate which elements of an array execute particular instructions and they are used to increase the flexibility of the content addressable memory. One unit of this three valued system is called a *trit*. Each trit can assume one of the values `0`, `1` or `X`. The binary operation we perform with trits is *equivalence*, which obeys the following "truth" table.

≡	0	1	X
0	1	0	1
1	0	1	1
X	1	1	1

The DSA architecture is capable of performing five basic instructions: **select**, **write**, **match**, **operate** and **readout**. The **select** instruction specifies which processing elements participate in the next sequence of instructions by writing a trit string into *SelectWord* of the DSA. Until the next **select** instruction, only lines whose address matches the contents of *SelectWord* perform any DSA operation. Thus, to make all lines active, *SelectWord* is filled with `X`'s. To make the even lines participate, a `XXXX ··· X0` is used, and if 3, 11, 19 and 27 are to participate, the select word will contain `0 ··· 0XX011`.

The other four instructions are executed in parallel by each of the selected processing elements in a SIMD fashion. The **write** instruction is used to write data into the CAM word of the selected processing element(s). Since *SelectWord* can contain `X`'s a **write** instruction can cause the CAM of more than one PE to be modified.

The **match** instruction includes a data word that is matched against the contents of the CAM words of each of the selected PE's using the equivalence function given above. The result of the match is then written into the *MatchLatch* where it can use used by the **operate** instruction. No data is transmitted out of the DSA by a **match** instruction.

The **operate** instruction causes each selected PE to perform a boolean operation on the contents of two registers and store the result in a third. This is a three operand instruction. As shown in Figure 1, one operand can come from the *MatchLatch* or a register of the adjacent PE's. The result of the boolean operation is also latched by the priority encoder.

The contents of the priority encoder are read using the **readout** instruction. The **readout** instruction returns the address of one of the lines whose priority encoder latch is set. At the same time it clears that particular priority encoder latch. Consequently, successive **readout** instructions return the addresses of the lines whose priority encoders contain a one. A special code is returned if all priority encoder latches contain zeroes.

2.1 Virtual Data Structure Accelerators

One of the most important features of using RAM to implement data structures is how little its performance degrades when the desired structure does not precisely match the underlying hardware. This section discusses the cost of a mismatch with a one dimensional DSA. The discussion for 2 and higher dimensions is similar, but substantially more complicated. RAM arrays have two parameters, the number of words in the system (ℓ) and the width of each word (n). If the desired data structure requires more than n bits per entry then several sequential words of memory can be used for each element. This slows down reading and writing to the data structure, but requires no hardware modifications. Alternatively, we can gang together several RAM systems (chips) to build a RAM system that has wider words than the component systems. This has higher bandwidth. If we want a RAM array with more words, we merely combine several RAMs using a selector to control access. Thus a sufficiently large RAM can emulate a smaller RAM with different parameters and, a worst, with linear performance degradation.

A common complaint about content addressable memories is that they don't easily scale in both dimensions—one has no recourse if the elements of the data structure are larger than the width of the content addressable memory. The data structure accelerator does not have this problem. A sufficiently large physical DSA can simulate a virtual DSA with any set of parameters. Furthermore, this simulation only imposes a linear penalty in time and lines used. This section demonstrates this simulation.

We will let V denote the *virtual* DSA we want to simulate using D, a real DSA with parameters (m, n). The basic idea of the simulation is fairly simple. Multiple lines of D are used to emulate a single line of V. There are three cases to consider: (1) simulating a (m, kn) DSA using a (m, n) DSA, (2) simulating a (km, n) DSA using a (m, n) DSA and (3) simulating a (m, n) DSA using a $(0, n)$ DSA. *It is possible to emulate CAM using RAM*, but not conversely because the CAM in the DSA cannot be written from the function generators on each line.

We begin by simulating a (m, kn) DSA with a (m, n) DSA. The only instruction that is affected by this simulation is the **operate** instruction. Rather than requiring 1 cycle, the simulation will require k cycles, where all but one of the cycles will be used to move the operands and/or the result into position. For the **operate** instruction, we only care about the contents of the register arrays after a sequence of operations. In particular, we don't care which function generator actually computes the result.

There are two cases when emulating the **operate** instruction: (1) when the destination register is on a line between the operands and (2) when it is outside the two operands. In the first case the boolean operation takes place in the same line as the destination. Figure 2 illustrates this. Without loss of generality, we assume the two operands are at lines 0 and $k-1$, and that the result of the operation is to be placed in line s. We move the first operand to line s, which takes $k-s-1$ cycles and the second operand to line s, which takes an additional $s-1$ cycles. The boolean operation is then performed, and the result is stored in the destination, line s. Thus at most $k-1$ cycles are required to emulate the (m, kn) **operate** instruction using an (m, n) DSA.

Simulating larger amounts of content addressable memory is also not difficult. Here the basic idea is to perform wide matches m trits at a time, and then use the function

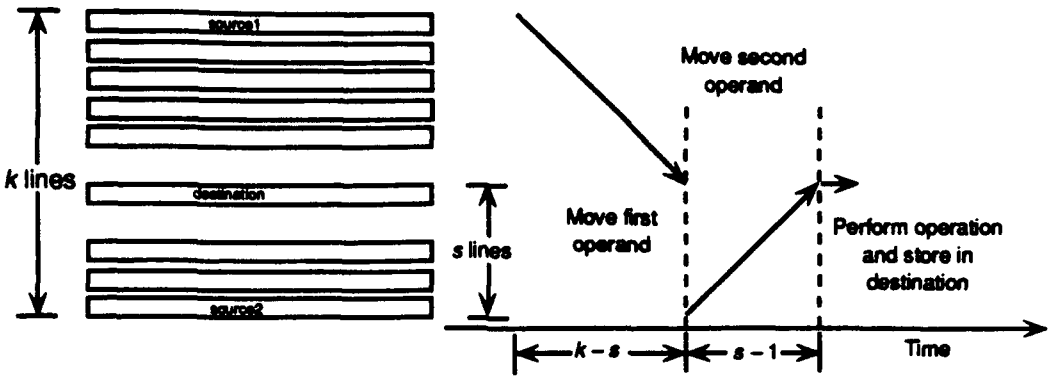

Figure 2: Multi-Line Operate Instruction

generators to combine the results. The following code illustrates this by simulating a `match` instruction on a $(4m, n)$ DSA using a (m, n) DSA. Each $4m$ trit line of the virtual DSA is implemented as four consecutive lines of the (m, n) DSA. We will assume the lowest numbered one consists of an even and an odd word. We use register 0 of the even line of the physical DSA as the virtual match latch.

```
QuadMatch(ArraySelector, WordZero, WordOne, WordTwo, WordThree) {
    select XX..X11 ∧ ArraySelector
       match WordThree
       R_i[0] ← M_i
    select XX..X10 ∧ ArraySelector
       match WordTwo
       R_i[0] ← R_{i+1}[0] ∧ M_i
    select XX..X01 ∧ ArraySelector
       match WordOne
       R_i[0] ← R_{i+1}[0] ∧ M_i
    select XX..X00 ∧ ArraySelector
       match WordZero
       R_i[0] ← R_{i+1}[0] ∧ M_i
    }
```

Each line in this code is precisely one DSA operation. We have written the operate instructions using an infix syntax for clarity. $R_i[0]$ denotes the 0 bit of the register set of the i^{th} DSA line. We use M_i to refer to the *MatchLatch*. Notice that the *SelectWord* is changed every third instruction, so the references to M_i and R_i are different in each block. In Section 3 we present a higher level, infix language that is much easier to use.

Using this technique we can use the DSA system to emulate virtual DSA arrays of arbitrary CAM widths, albeit with some performance loss. When the size of the virtual CAM word matches the physical CAM one virtual match instruction takes one physical match cycle. When the virtual word is n times the size of the physical CAM word, one virtual match instruction takes n physical match cycles plus $n-1$ operate cycles to combine

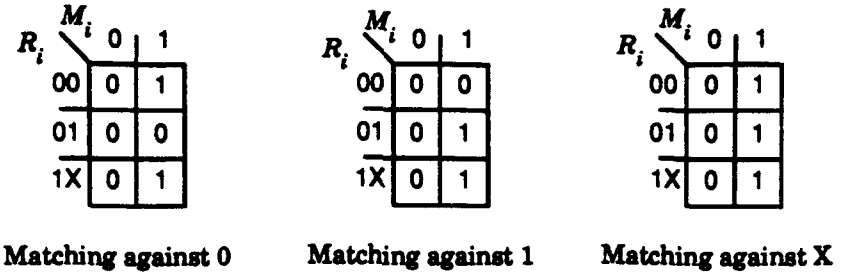

Figure 3: Match Simulation Truth Tables

the results of the partial matches. This is about a factor of 2 worse than the best that could be expected.

It is also possible to emulate the CAM completely using RAM. We can choose to emulate either a full, three level CAM or simpler two level CAM if that is all the application needs. In either case, the match emulation is performed one bit at a time. The following paragraphs illustrate the case of a trit based CAM word that is matched against a trit based match word. The resulting operations are simplified if one or both of the fields involved in binary.

We will choose one register to serve as the match latch and denote it by M_i. The firs. $2m$ registers will be used to hold the contents of the virtual CAM. The k^{th} bit of the virtual CAM will be stored in registers $R_i[2k]$ and $R_i[2k+1]$. If $R_i[2k]$ contains a 1 the k^{th} position of the CAM is an X, otherwise it contains the contents of $R_i[2k+1]$. Matches are performed by examining each trit of the match word and, depending upon its value, performing one of the boolean operations in Figure 3. That is,

$$M_i \leftarrow \begin{cases} M_i \wedge (R_i[2k] \vee \overline{R_i[2k+1]}) & \text{if matching a 0} \\ M_i \wedge (R_i[2k] \vee R_i[2k+1]) & \text{if matching a 1} \\ M_i & \text{if matching a X} \end{cases}$$

For each non-X in the match word two operate cycles are required. No operation need be performed for X's. Thus simulating a (m, n) DSA using a $(0, n)$ DSA will slow down match cycles by a factor of $2m - 1$.

All of these simulation results are summarized in the following table. The columns of the following table indicate the number of instructions required to perform an operation of a virtual $(km, \ell n)$ DSA on a DSA with the indicated parameters.

Operation	$(km, \ell n)$	(m, n)	$(0, \ell n)$
match	1	$2k-1$	$2km-1$
operate	1	ℓ	ℓ
write	1	j	km

2.2 Implementation Approaches

A data structure accelerator chip does not fall into any common class of integrated circuits. Because we want very large DSA arrays and the structure of the DSA is very regular,

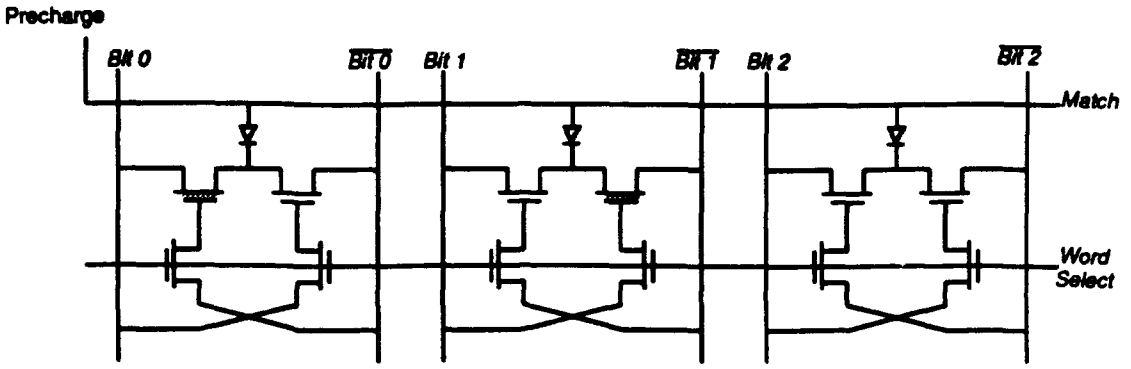

Figure 4: CAM cells

memory design techniques are needed. However, the DSA includes a significant amount of combinational logic (in the finite state machine) and arithmetic type circuits (in the priority encoder).

The DSA architecture may incorporate content addressable memory, which has the reputation of being difficult design an relatively space inefficient. Wade and Sodini have developed a cell (Figure 4)that is reasonable dense, using only 5 transistors, and relatively easy to design [7]. Using 2-μm design rules, buried contacts, single level metal and low resistance polycide lines gives a CAM cell with area of 25×22 μm^2. So the cost of this type of CAM cell is in the same ball park as a static RAM.

The first data structure accelerator implemented was the MIT database accelerator design [6] which is a (32, 4) DSA and used the Wade-Sodini cell. The ratio of eight between the CAM and RAM size was chosen because we expected the number of matches to significantly exceed the number of boolean operations. For many of the problems being considering, we would prefer more RAM per line of the DSA and can afford the the match time slow down inherent in a smaller content addressable memory. Thus we have been investigating a data structure accelerator that uses no CAM at all, but has a large number of RAM cells per line.[3]

Figure 5 shows this alternative implementation. It is much simpler because it does not contain a CAM. This eliminates the need for the `match` and `write` instructions, simplifying the control logic significantly. We have added extra sense amplifiers to latch one of the inputs and outputs of the function generator. The address buffers that control the row select logic need to be modified slightly to deal with trit addresses, but little else needs to be changed. This extra logic would be rather difficult to fit into a normal RAM row pitch, but the density should be quite a bit greater than that of the design shown in Figure 1.

The finite state machine of the DSA has been extended to be the entire processing element. Matches are much slower now, requiring 32 cycles if K_i contains a 32-bit quantity and 64 if it contains a 32-trit quantity. However, this architecture is a bit more flexible than the one shown in Figure 1 since the contents of the CAM cells can be modified by the

[3]This idea was originally suggested by Mark G. Johnson.

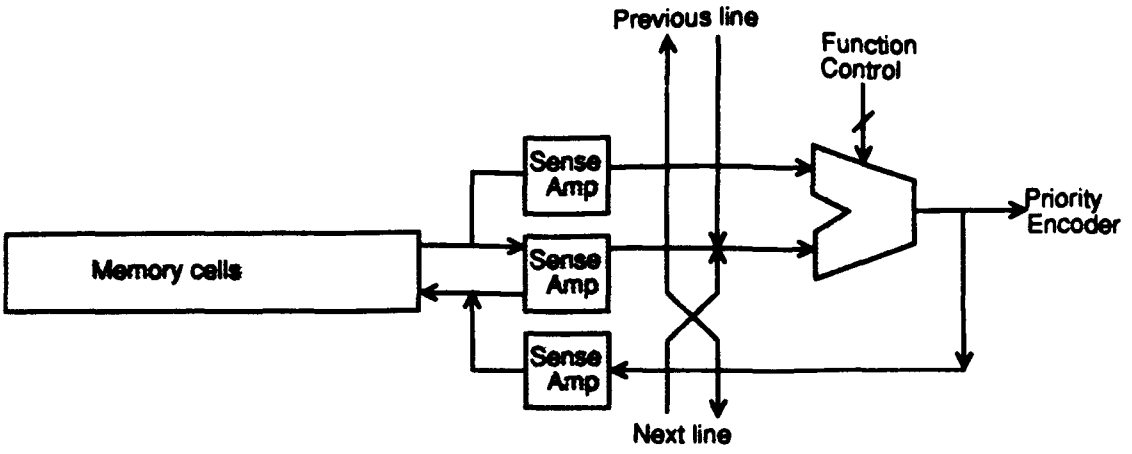

Figure 5: DRAM Approach to the Data Structure Accelerator

finite state machine.

We are currently designing a chip with this basic architecture with 128 RAM cells per line and we are trying to fit 10^3 lines on a single chip using a 1.25μm technology. The RAM cells are single transistor DRAM cells. We are building two variants of this design, one with a linear interconnect and second with a two dimensional interconnect for use in image processing applications.

3 An Algebraic Language for Specifying DSA Operations

We do not believe the DSA should be viewed as a universal computing engine but rather as a component of a heterogeneous computing system, where the DSA is used as a slave of some host processor or or perhaps shared for among several processors. The host is responsible for sequencing the DSA instructions and performing those data operations at which a SISD or MIMD machine would be preferable.

Algorithms that utilize the DSA are a mixture of DSA instructions and conventional single sequence processor instructions. Rather than expressing these algorithms in a mixture of low level DSA instructions and some high level language for the single sequence portion of the instruction stream, we have developed a set of high level extensions to a conventional block oriented programming language which allow the facilities of the DSA to be used effectively. This approach intertwines DSA operations with more conventional programming mechanisms including multiprocessing extensions. Rather than giving a complete description of the language, which is still evolving, this section describes the major features and provides enough information to make the examples in the later section clear.

The DSA description language have five basic components:

- Declarations that describe the allocation of DSA lines to different DSA arrays, and the allocation of CAM and RAM of the lines of a DSA array to various tasks.

- Basic operations for comparing the CAM contents with fixed data and performing boolean and arithmetic operations with the contents of the RAM.
- Loop abstractions that cause operations to be performed on blocks of DSA lines.
- A mechanism for describing algorithms best expressed as state transition tables.
- A library of higher level functions.

Each of these components is described in one of the following subsections. It is important to notice that our language intersperses DSA operations and conventional SISD operations. We believe this allows our description language to be more expressive, and properly leaves to the compiler the problems of the separating the operations that are performed on the DSA form those performed on the host processor.

3.1 Declarations

The N processing elements in a DSA are identified by their coordinates within the interconnection grid. These coordinates are used as a subscript. For one dimensional DSA's this is just an integer from 0 to $N-1$. Higher dimensional arrays use vector subscripts.

For instance, the RAM of the $\vec{i}^{th}$ line is written as $R_{\vec{i}}$, while the CAM of each line is written as $K_{\vec{i}}$. In the case of a one dimensional DSA we denote the RAM by R_i and the CAM by K_i. In the two dimensional case we use $R_{i,j}$ and $K_{i,j}$. The individual bits of the RAM can be referenced by $R_{\vec{i}}[0], \ldots, R_{\vec{i}}[n]$. The R_i, K_i and M_i registers are the hardware registers of a DSA and usually are not used directly by the programmer. Instead the programmer declares variables that are allocated from the available hardware resources using the declarations given below. This insulates the programmer from the complications of using the "virtualization" techniques described in Section 2.1.

Collections of DSA lines are called *DSA arrays*. Individual DSA arrays are allocated as if they were arrays, but whose elements are declared using `DSAstruct`. The purpose of the `DSAstruct` is to indicate to the compiler the RAM and CAM requirements of each line. For instance,

```
DSAstruct interval {
   CAM color[5];
   RAM selected, min[16], max[16];
   States S ∈ {inside, outside, unknown};
   }
```

defines the structure of a line of a DSA array. Only one CAM variable is allocated, `color`, which is 5 bits long. Three RAM variables are allocated, two of 16 bits and one of 1 bit. The `States` declaration indicates the allowable states of each line when programming the DSA using state transition techniques. The state transition techniques and the `States` declaration are described fully in Section 3.4. A DSA line with this structure will have at least 5 bits of CAM and 35 bits of RAM (one bit for `selected`, 16 each for `min` and `max` and 2 for `S`). Sometimes it is useful to have one variable overlay another. This is accomplished using the `Alias` declaration. Consider the following fragment:

```
RAM M[3], N[3], L[4] = Alias(M[1], N[2], M[0], N[0]);
```

If the compiler assigns M and N to the first six registers $R[0], \ldots, R[5]$ then L_i would be assigned as shown in the following diagram.

M_i			N_i		
$R_i[0]$	$R_i[1]$	$R_i[2]$	$R_i[3]$	$R_i[4]$	$R_i[5]$
$L_i[2]$	$L_i[0]$		$L_i[3]$		$L_i[1]$

DSA arrays are collections of one or more ***primitive blocks***, where each primitive block is a set of 2^k DSA lines on a 2^k boundary. Blocks of DSA lines are only allocated in sizes that are a power of 2 because of the organization of the decoders. Odd sized DSA array's are allocated as sets of primitive blocks. Primitive blocks are identified by the selector word that spans their elements. Thus $\mathtt{100XXX}_2$ represents the primitive block that extends from lines 32 through 39, inclusive. To indicate that the index i lies within this primitive block, we write $i \in \mathtt{100XXX}_2$.

A DSA array with 100_{10} elements would consist of primitive blocks of size 64, 32 and 4. It would be represented by the union of the identifiers for its constituent primitive blocks. For instance, for the DSA array defined by the statement:

```
DSAstruct interval Table[100];
```

we would have

$$\mathtt{Table} = \mathtt{100XXXXXX}_2 \cup \mathtt{1010XXXXX}_2 \cup \mathtt{1110000XX}_2$$

The lines of `Table` each contain at least 5 bits of CAM and 35 bits of RAM. We use a similar syntax to C to reference entries in DSA arrays. Subscripts are used to identify lines, so the fifth line of `Table`, all 40 bits of it are referred to as $\mathtt{Table}_5$. Particular variables are referred to concatenating the DSA array name with the variable name, separated by a slot, *e.g.* `Table.min` or $\mathtt{Table.min}_i$.

Set intersection and complement can also be used to describe DSA arrays. For instance, the even lines of `Table` might be denoted by

$$\begin{aligned}\mathtt{Table} \cap \mathtt{XXXXXXXX0}_2 &= (\mathtt{100XXXXXX}_2 \cup \mathtt{1010XXXXX}_2 \cup \mathtt{1110000XX}_2) \cap \mathtt{XXXXXXXX0}_2 \\ &= \mathtt{100XXXXX0}_2 \cup \mathtt{1010XXXX0}_2 \cup \mathtt{1110000X0}_2\end{aligned}$$

and the lines whose indices are not multiples of 4 by

$$\begin{aligned}&\mathtt{Table} \cap \overline{\mathtt{XXXXXXX00}_2} \\ &\quad = (\mathtt{100XXXXXX}_2 \cup \mathtt{1010XXXXX}_2 \cup \mathtt{1110000XX}_2) \cap (\mathtt{XXXXXXX1X}_2 \cup \mathtt{XXXXXXX01}_2) \\ &\quad = \mathtt{100XXXX0X}_2 \cup \mathtt{100XXXX01}_2 \cup \mathtt{1010XXX1X}_2 \cup \mathtt{1010XXX01}_2 \\ &\qquad \cup \mathtt{11100001X}_2 \cup \mathtt{111000001}_2\end{aligned}$$

Each of these three operations can be performed formally on the selector bit strings as follows. We consider the union of two selector bit strings $R = r_1 \cdots r_k$ and $S = s_1 \cdots s_k$. Assume R and S differ in just one bit position, so $r_i = s_i$ for $i \neq \ell$. There are then three possibilities:

$$R \cup S = \begin{cases} R & \text{if } r_\ell = \mathtt{X} \\ S & \text{if } s_\ell = \mathtt{X} \\ r_1 \cdots r_{\ell-1} \mathtt{X} r_{\ell+1} \cdots r_k & \text{otherwise} \end{cases}$$

The intersection of R and S can be performed on bit by bit basis. Assume r_i and s_i differ. If neither is an X then the intersection of R and S is the empty set. Otherwise, the intersection uses the bit which is not equal to X.

Complements are slightly tricky. The complement of R is a union of ℓ bit strings, where ℓ is the number of 0's and 1's in R. To see this examine the simple case $R = \overline{\texttt{X000}_2}$. We can trivially write R as a union of 7 bit strings, where each has one X in the same position. These strings are listed in on the left hand side of the double bars in the table below. On the right hand side, are given the three simplified bit strings whose union is R.

$\texttt{X100}_2$	$\texttt{X101}_2$	$\texttt{X110}_2$	$\texttt{X111}_2$	$\texttt{X1XX}_2$
$\texttt{X010}_2$	$\texttt{X011}_2$			$\texttt{X01X}_2$
$\texttt{X001}_2$				$\texttt{X001}_2$

These rules allow us to reduce all combinations of unions, intersections and complements of selector bit strings to unions of bit strings, *i.e.*, conjunctive normal form.

3.2 Basic Operations

Boolean and arithmetic operations with arbitrary sized RAM variables can be implemented in a bit serial fashion using the function generator. Thus we permit RAM variables to be combined using any of the standard boolean and arithmetic operations, provided their lengths are compatible. The following table gives the operators we currently use.

$\overline{A}$	logical negation
$\vee$	logical "or"
$\wedge$	logical "and"
$\oplus$	logical "xor"
$+$	addition
$-$	subtraction
$\leftarrow$	assignment

Consider the following code sequence:

```
DSAstruct Sample {
   RAM A[3], B[2], C[2];
   } S;
```
$S.A_i \leftarrow S.B_i + S.C_i;$

The compiler might allocate the variables A, B and C in RAM as

A_i			B_i		C_i	
$R_i[0]$	$R_i[1]$	$R_i[2]$	$R_i[3]$	$R_i[4]$	$R_i[5]$	$R_i[6]$

Then the statement $A_i \leftarrow B_i + C_i$ would be expanded into code equivalent to

$R_i[0] \leftarrow R_i[3] \oplus R_i[5];$
$R_i[\texttt{carry}] \leftarrow R_i[3] \wedge R_i[5];$
$R_i[1] \leftarrow R_i[4] \oplus R_i[6];$
$R_i[\texttt{carry}] \leftarrow (R_i[4] \wedge R_i[6]) \vee (R_i[4] \wedge R_i[\texttt{carry}]) \vee (R_i[6] \wedge R_i[\texttt{carry}]);$
$R_i[2] \leftarrow R_i[\texttt{carry}];$

3.3 Selection and Querying

Groups of instructions encapsulated in a loop-like block structure are used to indicate that a set of operations should be performed by a number of processing element. The syntax of these instruction is as follows:

```
ForEach i, ⟨boolean expression in i⟩ {
    ⟨forms involving i⟩
    }
```

The body forms are performed for each i that satisfies the boolean predicate. The simplest boolean predicate just indicates that i is an element of a particular set. For instance, the following code segment performs the previous operations on the 32 even lines in the range 0 to 63.

```
ForEach i, i ∈ 0XXXXX0₂
    Rᵢ[0] ← (Rᵢ[0] ∧ ¬Rᵢ[1]) ∨ Rᵢ[2]
```

This particular code segment will be expanded into a **select** instruction to set the *Select-Word* to $\texttt{0XXXXX0}_2$ and a few operate instructions for the body of the loop, *viz.*

```
select 0XXXXX0₂
Rᵢ[temp] ← Rᵢ[0] ∧ ¬Rᵢ[1]
Rᵢ[0] ← Rᵢ[temp] ∨ Rᵢ[2]
```

Odd sized DSA arrays, like the 100 entry **Table** given in Section 3.1, are dealt with by generating a DSA descriptor that is a union of selector bit strings. Then the body of the loop is repeated for each bit string. For instance, the sequence

```
ForEach i, i ∈ 100XXXXXX₂ ∪ 1010XXXXX₂ ∪ 1110000XX₂ {
    ⟨forms involving i⟩
    }
```

would be treated as:

```
ForEach i, i ∈ 100XXXXXX₂ {
    ⟨forms involving i⟩
    }
ForEach i, i ∈ 1010XXXXX₂ {
    ⟨forms involving i⟩
    }
ForEach i, i ∈ 1110000XX₂ {
    ⟨forms involving i⟩
    }
```

Notice that even though a **ForEach** loop evaluates its body at each line of the DSA array, the time required for the loop is typically $O(1)$, where the constant of proportionality is the time required to perform the body once. In the worst case, where the size of the DSA array is close to a power of 2, the loop will be performed $O(\log N)$ times for a DSA array with N lines.

Consider the following chunk of code:

```
ForEach i, (i ∈ 0XXXXX0₂) ∧ (Kᵢ ≡ Test) {
    Rᵢ[1] = Rᵢ[1] ∧ Rᵢ[2];
    printf ("Line %d matched", i);
    }
```

The predicate for this loop is a bit more complex. The body is performed for each of the even lines in the range 0 to 63 whose CAM's contents match **Test**. This predicate is expanded into three instructions: a **select** instruction that sets the *SelectWord* and a **match** instruction for $K_i \equiv$ **Test**. In addition the result of this match is stored in a register (R_i[loop]) for later use.

The first statement in the body is a simple **operate** cycle, except that it is only supposed to take effect on those lines that R_i[loop] = 1. This is accomplished by conditionalizing writes in the loop on the value of R_i[loop]. Thus the first line of the body expands into:

$$R_i[1] = (R_i[\text{loop}] \wedge R_i[1] \wedge R_i[2]) \vee (\overline{R_i[\text{loop}]} \wedge R_i[1]);$$

The final statement in the body actually expands into a loop. First an **operate** instruction issued to store the contents of the R_i[loop] in the priority encoder register. Then a **readout** instruction occurs for each of the designated lines, followed by the code for the **printf** statement which uses the value returned by the successful **readout** instructions.

The set predicates available for use in a **ForEach** statement include multibit tests, multiple matches and arithmetic comparisons. The arithmetic comparisons are discussed in Section 3.5.

3.4 State Transitions

A common approach to using the processing elements of the DSA is as a state machine. To make this a bit easier for the programmer and to make the resulting programs a bit more intelligible, we have decided to have the compiler allocate the binary patterns for states and work out the state transition equations. This is accomplished with two new types of statements.

A new set of states is introduced by the **States** form,

```
States S ∈ {up, down, sideways};
```

This statement declares S to be a *state identifier* for each PE. The state of any particular PE is indicated by adding a subscript. Thus the state of the 5^{th} processor is S_5. If we wanted to find all the processing elements which are in the *up* state, we would use the following code segment:

```
ForEach i, Sᵢ = up
    printf ("Line %d is up, i);
```

Occasionally, computations may involve more than one set of orthogonal states. In this case, several state variables are declared, as in the following example.

```
States S ∈ {up, down, sideways};
States T ∈ {red, yellow, blue};
```

With these declarations, each PE could be in one of nine different states. We also say that the *S-state* of a processing element is S_i and that its *T-state* is T_i.

States can be changed by using the **NewState** form. The **NewState** form identifies which state variable is to be changed and has body consisting of a set clauses indicating how to change the state. For instance, we might have

```
States S ∈ {up, down, sideways};
ForEach i, i ∈ "XX...XX" {
   NewState S {
      up:   S_i ← down;
      down:  if K_i ≡ "X...X11X"
                then S_i ← up;
                else S_i ← sideways;
      otherwise:  S_i ← sideways;
      }
   }
```

If the compiler chose to use registers $R_i[0]$ and $R_i[1]$ to hold the state of each processing element, as follows:

State	$R_i[0]$	$R_i[1]$
up	0	0
down	0	1
sideways	1	X

There are three different binary inputs to the truth table for this state transition, the two state variables $R_i[0]$ and $R_i[0]$ and the result of the match $K_i \equiv$ "X...X11X", which we denote by M_i. This gives the following Karnough map:

	00	01	11	10
M_i	01	00	1X	1X
$\overline{M_i}$	01	1X	1X	1X

To minimize the the number of produce terms we use the following assignment of the X's.

	00	01	11	10
M_i	01	00	10	11
$\overline{M_i}$	01	10	10	11

Thus the original state transition code is equivalent to

```
ForEach i, i ∈ "XX...XX" {
   R_i[0] ← ((K_i ≢ "X...X11X") ∧ R_i[0]) ∨ R_i[0];
   R_i[1] ← ¬R_i[1];
   }
```

which, though somewhat shorter, is significantly less clear than the state transition code given earlier.

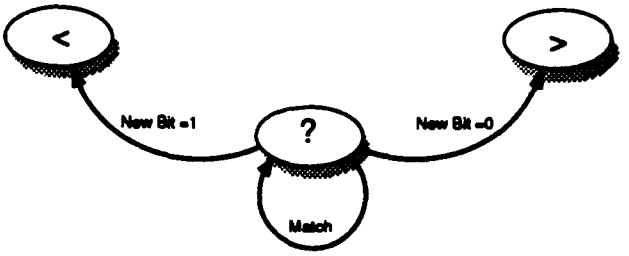

Figure 6: State Diagram for Comparison

3.5 Arithmetic Comparisons

One of the fundamental applications of the data structure accelerator is searching tables quickly. The content addressable memory accelerates searches involving boolean patterns while the function generator accelerates searches involving arithmetic patterns. For example, we can determine those lines of a DSA that contain a number of lying between given bounds, or the line of the DSA that contains the largest quantity in time independent of the number of lines being searched.

For simplicity we assume that each line of the DSA contains precisely one key and the key is is m bits long. This key can lie either in the CAM portion of the DSA or in the RAM portion. In this section we assume the key lies in the CAM, but trivial modifications of the algorithms enable it work with the key in the RAM. We consider two fundamental operations, comparison with a given quantity and finding the largest element of a set of keys.

The following function identifies each line whose key is greater than **LowerBound**. This is done by examining each of the bits of the key in sequence. Each DSA line can be one of three states: *greater*, *lesser* and *unknown*. In state *lesser* the entry is known to be less than the key; in the *greater* state it is known to be greater than the key, and in state *unknown* we still don't know. If an entry is in state *unknown* then its leading bits match the leading bits of the key that have been presented so far.

All words are initially placed in the *unknown* state. We compare the contents of the CAM with **LowerBound** one bit at a time, changing the state of the line as necessary. The state transition diagram is shown in Figure 6. At the end of m cycles, any word still in state *unknown* is equal to the key.

The following program uses two special arrays. **BitMask**$[n]$ contains a 1 in the n^{th} bit position, from highest to lowest. Thus **LowerBound** $\wedge$ **BitMask**$[n]$ selects the n^{th} bit from **LowerBound**. **FieldMask** is similar, but has the n highest bits set.

```
Compare(Array, LowerBound) {
   States Array.S ∈ {greater, lesser, unknown};
   ForEach i, i ∈ Array {
      Array.S_i ← unknown;
      for 0 ≤ n < MatchWidth {
          NewState S {
             unknown:  if Array.K_i ≡ (LowerBound ∧ FieldMask[n])
```

```
                    then Array.S_i ← unknown;
                    else if 0 = LowerBound ∧ BitMask[n]
                            then Array.S_i ← greater;
                            else Array.S_i ← lesser;
            otherwise:  Array.S_i ← Array.S_i;
            }
        }
    }
}
```

This and similar routines are used by the compiler to implement arithmetic predicates in **ForEach** statements. For instance, one might want to use the DSA to represent a large set of one dimension intervals. The following code segment would then be used to find those intervals that contain the origin.

```
DSAstruct Sample {
   CAM L[16], R[16];
   } S;

ForEach i, (S.L_i < 0) ∧ (S.R_i > 0)
   printf("Interval %d contains the origin.", i);
```

For example, we might want to use the DSA to represent a set of intervals from 0 to $2^{16} - 1$. Each interval would be represented as one line of a $(32, n)$ DSA, with half of K_i used to hold the lower limit of the interval and half for the upper limit. A modest sized data structure accelerator could then contain a rather large set of these intervals. Using this algorithm, we can determine the intervals in this set that intersect a given interval in time linear in the size of the intersection. This technique trivially extends to two or more dimensions.

3.6 Extremum Searches

In this section we consider two different types of extremum searches: finding the largest element in a DSA and finding the closest match to a quantity using the Hamming distance. These problems can be described as follows. Assume we are given a set of n keys $\mathcal{K} = \{K_r = k_{r0}k_{r1} \wedge k_{rm}\}$, where the K_r are m bit binary numbers and $n \gg m$. The extremum problem is to determine the j such that K_j is the largest element of $\mathcal{K}$. In the Hamming distance problem we are to find the element of $\mathcal{K}$ that differs from a given m bit pattern P in the fewest number of positions.

The solutions to these problems all involve examining the K_i one bit a time. Thus each algorithm takes time linear in the length of the keys, but is independent of the number of keys. The keys for these algorithms can either be in the CAM or RAM portion of the DSA. An algorithm that expect the keys to be in the RAM can be syntactically transformed into one expects it to be in the CAM, *viz.* a sequence of matches against **1XXX**..., **X1XX**..., **XX1X**... moves the bits of the CAM into M_i one at a time, where a RAM based algorithms can used them. This adds an extra instruction for each bit of key width, and at worst slows the algorithm by a factor of 2.

The second case, where the key is in the CAM but the algorithm expects it to be in the RAM is only slightly more difficult. We can use the techniques of Section 2.1 to simulate the CAM using RAM. In the worst case this will slow down the algorithms by a factor of m. Often however, algorithms look at the keys one bit at a time, so only a few extra instructions are introduced.

We begin with the algorithms that find the largest key in $\mathcal{K}$. The basic technique is to determine the largest element, one bit at a time, starting from the highest order bit position. This is done by matching the DSA array against `1XXXX`.... If any lines match this pattern, then the largest key in $\mathcal{K}$ has its highest order bit equal to 1, other wise it is 0. There is a match, we continue with a match word of `11XXX`..., otherwise `01XXX`.... In either case we determine the two highest bits of the largest element of $\mathcal{K}$ in 2 cycles, and the largest key in m using match instructions. At most one additional match is required to load the priority encoder with the address of the largest key. The following code fragment implements this idea.

```
match ← "XXXX..XXX";
ForEach i, i ∈ Array {
   for m − 1 ≥ r ≥ 0 {
       match[r] ← 1;
       unless (Array.K_i ≡ match)
         match[r] ← 0;
       }
   } return(match)
```

Sometimes it is necessary to determine those entries that differ from a given pattern $key = k_0k_1k_2\ldots$, by only a few bits. This is useful in image processing an data correction applications. For concreteness assume the key and each DSA entry contains 32 trits. Determining which entries differ from *key* by a single bit can be done in 32 match cycles by match against a copy of *key* with a single `X` in each of the 32 different bit positions. When done in this fashion, two bit mismatch requires 1024 match cycles. This is not a very efficient approach when there is a large mismatch.

For large mismatches a faster and simpler approach is to check each bit position individually by matching against k_0`XXX`..., `X`k_1`XX`... and so on. We then increment a counter each time there is a mismatch. After 32 match cycles we will know the number of mismatches for each entry in the DSA.

It is easy to develop routines that determine the lines that mismatch a key at a given number of bit positions. For instance the following routine determines the lines of a DSA array that mismatch `word` in at most 3 bit positions.

```
HammingMatch (Array, word) {
   RAM Counter[n];
   ForEach i, i ∈ Array {
      Array.Counter ← 0;
      pattern ← "XXXXXXXXX..XX";
      for 0 ≤ j < m {
        pattern[j] ← word[j];
        Array.Counter ← Array.Counter + 1;
        pattern[j] ← "X"
```

```
        }
    }
}
```

There are a vast number of operations that can be implemented using the finite state machines int he data structure accelerator. For instance, the **Compare** routine can be easily modified to determine for which lines K_i is between specified upper and lower bounds. It is not difficult to find the line containing the largest element or that differs from a given quantity by the smallest Hamming distance. (This could be used for linear pattern matching.)

3.7 A Space/Time Tradeoff

One unfortunate effect of the linear nearest neighbor interccnnection network of the data structure accelerator is that the (graph theoretic) diameter of a data structure accelerator of n lines is n. Thus computations that requires data in distant lines be combined can be quite slow. The boolean n-cube type networks used by the Connection Machine [3] has a diameter of $\log n$ and thus can perform somewhat better with these algorithms. Unfortunately, boolean n-cube networks do not scale to large numbers of processing elements. In fact, networks with diameter less than $n^{1/3}$ cannot be embedded in 3-space in a uniformly scalable fashion. Power distribution and heat dissipation considerations raise this bound to $n^{1/2}$ and packaging considerations to n. Thus algorithms that require a high degree of communication among $O(n)$ processing elements will require more than $O(n)$ "real estate" in realizable systems.

In this section we show how to solve such a problem using $O(n^2)$ lines of a DSA. Notice that while fewer processors could be used if a smaller diameter network were used, it is not clear that less total hardware would be required.

Consider the following problem: Given a set of n integer $\{a_0, \ldots, a_{n-1}\}$, find those pairs that have the minimum difference. The brute force approach of comparing all pairs of integers requires $O(n^2)$ operations. However, this can be reduced to $O(n \log n)$ operations by first sorting the integers an then comparing their neighbors. Using a DSA we can solve this problem using $O(n)$ space and $O(n)$ time in the following fashion. First, store each a_i in a line of the DSA. Then, in parallel, compute the difference between the contents of each line and a_0. Repeat this for each a_j retaining the smallest difference. This will take $O(n)$ operations. Finally, the smallest difference is determined using the techniques of Section 3.6. This requires $O(1)$ operations. Thus using $O(n)$ lines of a DSA we can solve this problem in time $O(n)$.

An alternative approach is to store in each of n^2 lines of the DSA the pairs (a_i, a_j). Then the n^2 differences can be computed in parallel using $O(1)$ operations. This may be useful approach if many such calculations with the same set of a_i are to be performed. The only problem is to get the data into the DSA efficiently.

Observe that the n^2 entries in the DSA can be written using only $O(n)$ operations by using the selector carefully. Assume that $n = 2^k$, We begin by writting a_i in the n lines beginning with line in. Then write a_i in lines i, $n + i$, $2n + i$ and so on. Each of these two passes requires $O(n)$ write operations so the entire n^2 array can be set up in $O(n)$ time.

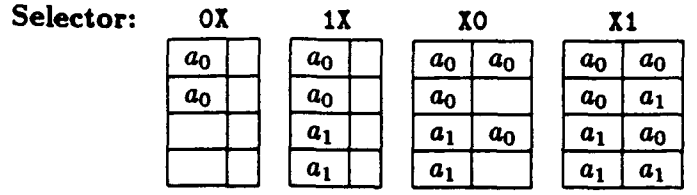

Figure 7: Intermediate states while creating a cross product

The following code fragment implements this procedure for a 4×4 array. For simplicity, we have (unrealistically) assumed that each of the a_i is a single bit. Notice that each line is a single DSA instruction. Figure 7 illustrates the operation of this technique when applied to 2×2 case.

```
ForEach i, i ∈ "00XX"
    Ri[left] ← a0;
ForEach i, i ∈ "01XX"
    Ri[left] ← a1;
ForEach i, i ∈ "10XX"
    Ri[left] ← a2;
ForEach i, i ∈ "11XX"
    Ri[left] ← a3;
ForEach i, i ∈ "XX00"
    Ri[right] ← a0;
ForEach i, i ∈ "XX01"
    Ri[right] ← a1;
ForEach i, i ∈ "XX10"
    Ri[right] ← a2;
ForEach i, i ∈ "XX11"
    Ri[right] ← a3;
```

4 Problems in Computational Geometry

In this section we demonstrate how the data structure can be used to solve several problems in computational geometry. Since the problems in computational geometry typically arise as a component of other applications (such as VLSI or mechanical CAD), we think the use of the data structure accelerator is particular appropriate. It is relatively inefficient to design specialized hardware to solve the computational geometry problems that arise in CAD, because they are just one component of a larger computational problem. And yet there is a huge amount of potential parallelism to be exploited. The data structure accelerator provides that parallelism in a fashion that is not specialized to the problems of computational geometry. At the same time, these techniques require using the data structure accelerator in tandem with regular processing elements. This is precisely the type of cooperative heterogeneous computation discussed in the introduction.

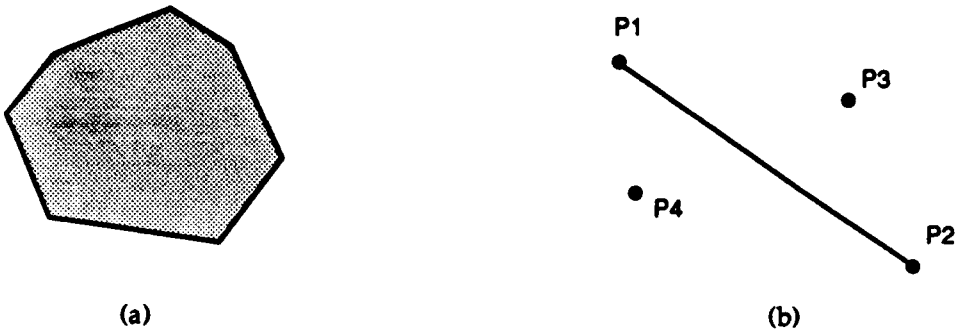

Figure 8: Polygon Inclusion

4.1 Convex Polygon Inclusion

The convex polygon inclusion problem is relatively straightforward. A convex polygon is described by the sequence of its vertices, as shown in Figure 8(a). We are to determine if a given point is contained within the polygon. The basic relationship we use is illustrated in Figure 8(b). If we denote the x and y coordinates of the point P_i by x_i and y_i, the signed area of the triangle $P_1P_2P_3$ is

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_2y_3 + x_3y_1 - x_1y_3 - x_2y_1 - x_3y_2.$$

The sign of the area is positive if the points P_1, P_2 and P_3 are arranged counterclockwise in the plane [5]. Thus in Figure 8(b),the triangle $P_1P_2P_3$ has positive area, while the triangle $P_1P_2P_4$ has negative area.

To determine if a point P is contained within a polygon we check that the triangles formed by P and each edge of the polygon have positive area. This is easily done by assigning each edge of the polygon to a line of the DSA:

```
DSAarray LineSegment {
   RAM Lx[ℓ], Ly[ℓ], Rx[ℓ], Ry[ℓ], ;
       Area[2ℓ];
   };
```

We have allocated four ℓ-bit quantities to hold the coordinates of the endpoints, and one 2ℓ-bit quantity for the area of the triangle in the computation.

```
PolygonInclusion(Edges, Px, Py)
DSAarray LineSegment Edges□; {
  ForEach i, i ∈ Edges {
     Edges.Area_i ←Edges.Lx_i × Edges.Ry_i + Edges.Rx_i × Py + Px × Edges.Ly_i
                  − Edges.Lx_i × Py_i − Edges.Rx_i × Edges.Ly_i − Px_i × Edges.Ry_i;
     if ∃j.(Edges.Area_j < 0)
        then return( "Outside");
```

```
        else return( "Inside");
    }
}
```

The time required by this algorithm is independent of the number of edges of the polygon(s), but due to the multiplications in the area computation, quadratic in the number of bits required to represent the coordinates of the vertices $O(\ell^2)$. We could say that the time is $O(\ell \log \ell)$ by using an FFT algorithm, but this would only be of theoretical interest and ignores the performance cost of getting a larger algorithm to the DSA. In addition, we must count the preprocessing time required to load the n vertices into the DSA, which is $O(\ell n)$, since each point has size $O(\ell)$. The time to check m points grows to $O(m\ell^2))$, while the preprocessing time remains fixed (using classical multiplication).

If the number of points is large, or the same points are used repeatedly with different sets of polygons, we can store the points the DSA and perform the computation for different polygons. An example of this typeof problem is given in Section 4.3. In this case the preprocessing time becomes $O(\ell m)$ while the computing time becomes $O(n\ell^2))$.

Classical algorithms [5] require $O(\ell n)$ time for preprocessing and answer the inclusion question for m points in time $O(\ell^2 m \log n)$. For comparison, these results are summarized below.

	Preprocessing	Query
vertices in DSA	ℓn	$m\ell^2$
points in DSA	ℓm	$n\ell^2$
classical	ℓn	$\ell^2 m \log n$

The DSA approach uses a straightforward algorithm and achieves somewhat better performance than the classical techniques. The following section discusses a variant of this problem where the test points lie in a regular grid.

4.2 Polygon Filling

A common primitive for a number of geometric algorithms is *polygon filling*. In the two dimensional case, we are given a rectangular section of the plane discretized into an $m \times n$ grid. Within this grid are marked the boundaries of a number of connected regions. The polygon filling problem is to identify the regions in which each point of the grid is contained.

A simple example of this problem arises in computer graphics where the regions might represent homogeneous regions of an image, *e.g.* surfaces of objects. When the image is presented on the screen, each pixel within each region needs to be painted with the same color.

Figure 9 shows a two dimensional region embedded in a grid. Each dot represents a single processing element of a two dimensional DSA. The diagonal lines form the true boundary of the region, while the black dots indicate the boundary on the grid. Notice that each black dot lies on or within the boundary of the region. Given such a boundary we can propagate a seed node outward until it reaches a boundary. This is illustrated in Figure 9 where the seed node is at $(6, 5)$. At t_0 it is the only node marked. Between t_i and t_{i+1} each marked node propagates a mark to each of its four neighbors if they are (1) not already marked and (2) not a element of the boundary. The time at which each node is marked is

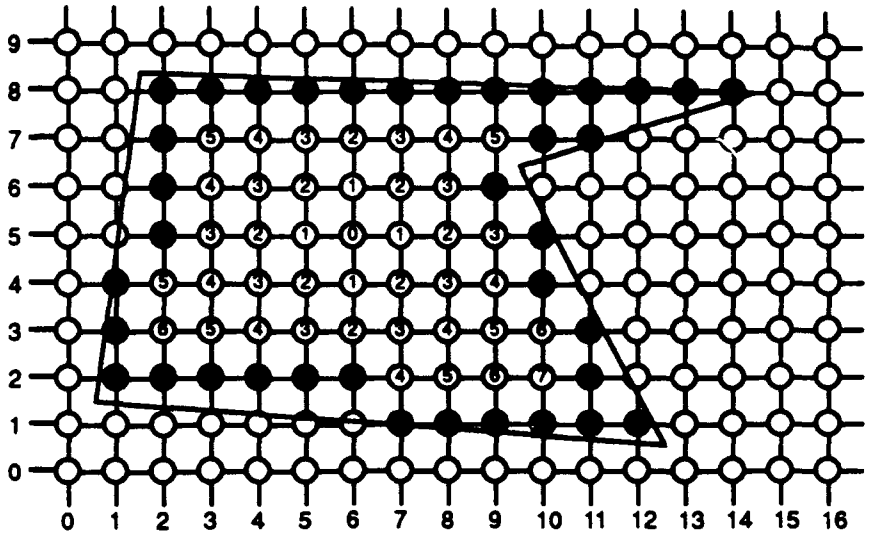

Figure 9: Sample Image

given in the figure. In this case every node in the region that is orthogonally connected to $(6,5)$ is marked in 7 units of time.

The data structure used to model the grid is defined as follows.

```
DSAstruct FillGrid {
   States S ∈ {interior, exterior, boundary, unknown};
   } Grid[n, n];
```

Each node in the grid can be in one of four states: *interior*, *exterior*, boundary and *unknown*. Initially each node is placed in the ***unknown*** state. The boundary is then defined by place each node on or just inside the boundary to the ***boundary*** state. The orientation of the boundary is defined by setting one node inside the region to the *interior* state. This node serves as a seed that spreads throughout the region.

The following block of code then propagates the seed throughout the interior.

```
Propogate (Grid) {
   for ℓ < n {
     ForEach (i, j), (i, j) ∈ Grid {
        NewState Grid.S {
           unknown:  if (Grid.S_{i+1,j} = interior) ∨ (Grid.S_{i,j+1} = interior)
                              ∨ (Grid.S_{i-1,j} = interior) ∨ (Grid.S_{i,j-1} = interior)
                        then Grid.S_{i,j} ← interior;
                        else Grid.S_{i,j} ← unknown;
           otherwise:  Grid.S_{i,j} ← Grid.S_{i,j};
        }
     }
```

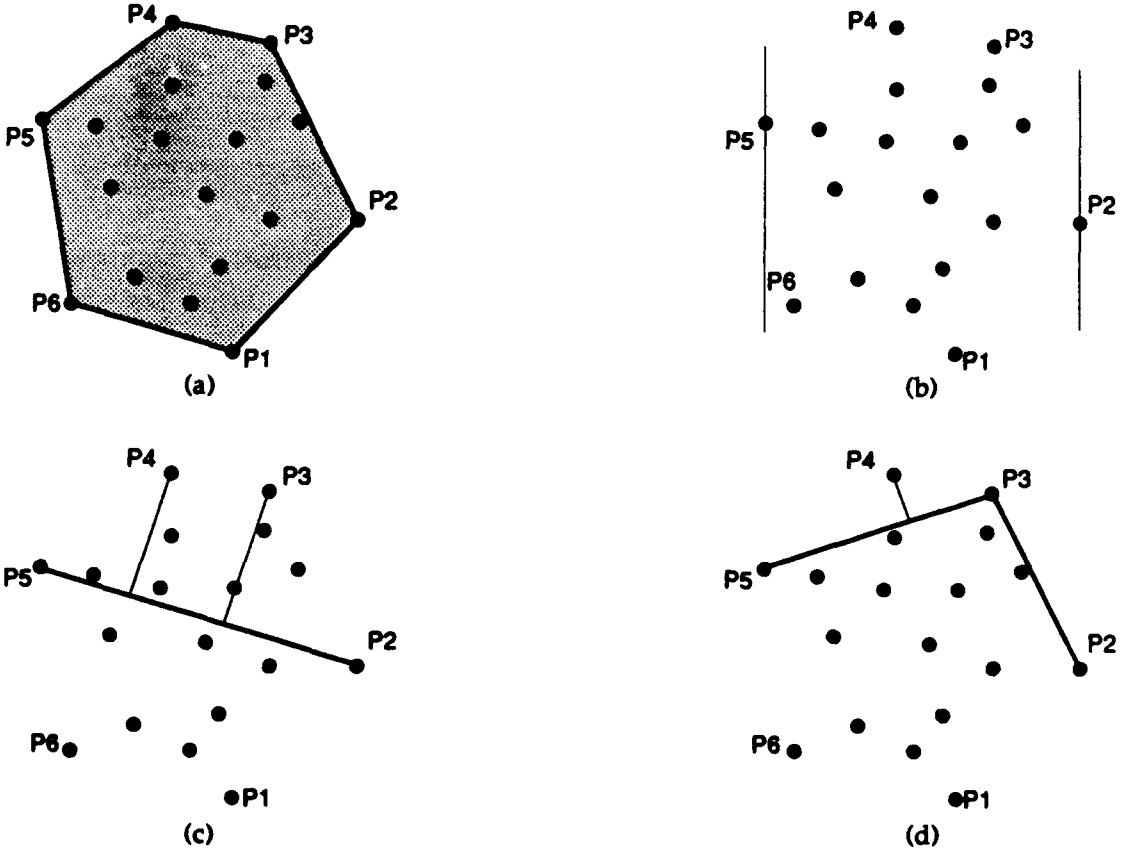

Figure 10: Convex Hull

```
    }
}
```

This routine assumes the orthogonal distance between any two connected nodes is no more than n. This is the case for convex polygons, but serpentine (concave) polygons can be created whose interiors have minimal orthogonal paths of length $O(n^2)$.

4.3 Convex Hull

The *convex hull* of a set of points $S = \{P_1, \ldots, P_n\}$ is the subset of S whose elements are the vertices of a convex polygon that includes all the points in S. This is illustrated in Figure 10(a). The black dots are the points in S, and the points that are on the vertices of the shaded polygon form the convex hull. The approach we will use to find the convex hull using the data structure accelerator is illustrated in Figure 10(b–d). We first enter each point into a DSA array, with one point per line. We then find a box that will bound the hull by finding the four points with the largest and smallest x and y coordinates. These four points form the beginning of the hull. As can be seen in Figure 10(c), some points

remain outside the hull.

We start with one vertex of our initial hull and walk along the edges connected to it, refining the edges as necessary to form the convex hull. For instance, assume we start with P_1 in Figure 10(c) and proceed counter clockwise. All of the points in S are on the same side of the line P_1P_2. Thus the edge $\overline{P_1P_2}$ is in the convex hull. Moving on to the edge $\overline{P_2P_4}$ we find three points on one side of the line (outside the current convex hull) and the rest on the other. One of these three points needs to be added to the set of vertices. The one to add is the one whose distance from the line $\overline{P_2P_4}$ is maximal, P_3. Adding this to our set we get the hull shown in Figure 10(d). We then continue with the edge $\overline{P_2P_3}$.

To determine if a point P is inside the edge $\overline{P_iP_j}$ we compute the signed area A_P of the triangle $\triangle P_iP_jP$. If we are proceeding around the hull in a counterclockwise fashion, the area must be positive to be inside this segment of the hull. (In Figure 10(c) P_6 is "inside" the segment $\overline{P_2P_4}$, but it is outside the hull.)

Among those points with negative area, we must find the one that is furthest from the segment $\overline{P_iP_j}$. The distance of a point P from the line $\overline{P_iP_j}$ can be determined by dividing the area of the triangle $\triangle P_iP_jP$ by twice the length of the line segment $\overline{P_iP_j}$. Given two points, P and Q, P will be further than Q from $\overline{P_iP_j}$ if

$$\frac{A_P}{2\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}} > \frac{A_Q}{2\sqrt{(x_i-x_j)^2+(y_i-y_j)^2}}.$$

So the point furthest from P_iP_j is the one whose triangle has the largest (unsigned) area.

The sample points are stored in the DSA using a structure similar to `LineSegment` used in Section 4.1:

```
DSAarray HullPoint {
   RAM X[k], Y[k], Area[2k];
   }
```

The DSA code for a simplified version of `ConvexHull` is show in Figure 11. This function takes an argument `P`, a DSA array whose lines are `HullPoints`. The first two `ForOne` blocks are used to determine the rightmost and leftmost points in `P`, respectively. The next statement adds the two lines $\overline{P_iP_j}$ and $\overline{P_jP_i}$ to the set of candidate lines.

We then remove a line $\overline{P_iP_j}$ from the candidate set and compute the maximum of the areas of the triangles by combining every point in `P` with the base $\overline{P_iP_j}$. If the maximum is not positive, then every point is either inside the edge, or lies on the edge. Thus the edge $\overline{P_iP_j}$ should be added to the convex hull. Otherwise we find a point with maximal (signed) distance from $\overline{P_iP_j}$, say P_k, and add the lines $\overline{P_iP_k}$ and $\overline{P_kP_j}$ to `Candidates`. The algorithm terminates when `Candidates` is empty.

The sum of the number of elements in `HullEdges` and `Candidates` never exceeds the number of edges in the final convex hull, which we assume to have h edges. Thus the number of operations required by `ConvexHull` is $O(h)$. It will take $O(n)$ operations to insert the the points in S into the DSA, so the entire process will require $O(h+n)$ operations. The space required by the algorithm is obviously $O(n)$.

The algorithm in Figure 11 makes a couple of simplifications that we might not in a real program. First, we've ignored the degenerate situation where all of the points of S fall on

```
ConvexHull (P)
DSAarray HullPoint P[]; {
   Point Pi, Pj;
   Integer MaxArea;
   Set HullEdges, Candidates;
   ForEach i, i ∈ P {
      ForOne j, P.X_j = max(P.X_i)
         Pi ← ⟨P.X_j, P.Y_j⟩ ;
      ForOne j, P.X_j = min(P.X_i)
         Pj ← ⟨P.X_j, P.Y_j⟩ ;
      Candidates ← Candidates ∪ (Pi, Pj) ∪ (Pj, Pi);
      for((Pi, Pj) ∈ Candidates) {
         P.Area_i ← (Pi.X · Pj.Y − Pj.X · Pi.Y) + (Pi.Y − Pj.Y)P.X_i + (Pj.X − Pi.X)P.Y_i;
         MaxArea ← max(P.Area_i);
         if (MaxArea ≤ 0)
            then HullEdges ← HullEdges ∪ (Pi, Pj);
            else
              ForOne k, P.Area_k = MaxArea
                 Candidates ← Candidates ∪ (Pi, ⟨P.X_k, P.Y_k⟩) ∪ (⟨P.X_k, P.Y_k⟩, Pj);
         }
      }
   }
```

Figure 11: Convex Hull Algorithm

a vertical line. In this case the two points, `Pi` and `Pj`, computed by the first two `ForOne` blocks could be identical. Second, there could be more than one point which is maximally distant from the $\overline{P_iP_j}$. In the final `ForOne` block we only pick one of these points to refine $\overline{P_iP_j}$. By using all of them we could reduce the number of refinement steps required, at the cost of using the sequential machine to sort the refinement points.

Finally, it should be noted that somewhat more complex variants of this algorithm can be applied to compute convex hulls of sets of points in higher dimensions. Almost exactly the same idea is used although care must be exercised in choosing the initial candidate planes to avoid degenerate cases. Again the time required by this algorithm will be $O(h+n)$ and $O(n)$ space will be needed.

5 Conclusions

In this paper we have discussed a massive SIMD architecture that is designed to be used as an integral component of a heterogeneous highly parallel machine. We have described a few basic algorithms for using the data structure accelerator and provided a language for describing other algorithms. One of the benefits we observe from the free use of SIMD technology is that many algorithms in computational geometry can be dramatically simplified, so that the straightforward algorithms appear to be as good as the sophisticated ones used on sequential machines. However, maximum advantage of the SIMD organization is obtained when then SIMD structures are combined with more conventional single sequence, or MIMD architectures. In this case the SIMD architecture proposed can be naturally embedded in the memory systems.

This paper benefited from the comments of Laurie Hendren, James Stewart and Steve Vavasis. Paul Chew corrected a number oversights and generally improved content and presentation of this paper.

References

[1] Sharon Marie Britton. 8k-trit Database Accelerator with error detection. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, February 1990.

[2] W. Daniel Hillis and Guy Lewis Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[3] W. Danny Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[4] Jerry L. Potter. *The Massively Parallel Processor.* MIT Press Series in Scientific Computation. MIT Press, Cambridge, MA, 1985.

[5] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry.* Texts and monographs in computer science. Springer-Verlag, New York, 1985.

[6] Jon P. Wade, Peter Osler, Richard E. Zippel, and Charles Sodini. The MIT Database Accelerator: 2k-trit circuit design. In *1987 Symposium on VLSI Circuits*, Karuizawa, Japan, January 1987.

[7] Jon P. Wade and Charles G. Sodini. Dynamic cross-coupled bitline content addressable memory cell for high density arrays. *IEEE Journal of Solid State Circuits*, SC–22(2):119–121, February 1987.

[8] Jon P. Wade and Charles G. Sodini. A ternary content addressable search engine. *IEEE Journal of Solid State Circuits*, SC–24(4):1003–1013, August 1989.

[9] C. Weems, S. Levitan, and Caxton Foster. Titanic: A VLSI based content addressable parallel array processor. In *Proceedings of 1982 International Conference on Custom Circuits*, pages 236–239. IEEE, 1982.